7-14-1994

Reliable General Purpose Dynamic Memory Management for Real

Hong Gao
Iowa State University

Kelvin Nilsen
Iowa State University

Reliable General Purpose Dynamic Memory Management for Real

Abstract
Traditional dynamic memory management techniques for imperative programming languages are unsuitable for reliable real-time applications because their worst-case time and space requirements are insufficiently bounded. This is demonstrated by detailed measurements of several real-world workloads. A special hardware-assisted real-time garbage collection system has been designed to facilitate reliable use of dynamic memory in hard real-time systems. By analyzing the dynamic memory use of application software, the real-time developer can prove compliance with time and space constraints. Analysis techniques are presented and the real-time performance of the hardware-assisted garbage collection system is compared to that of traditional allocators.

Disciplines
Systems Architecture

Reliable General Purpose Dynamic Memory Management for Real-Time Systems

TR94-09
Hong Gao, Kelvin Nilsen

July 14, 1994

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

Reliable General Purpose Dynamic Memory Management for Real-Time Systems

Hong Gao
Kelvin D. Nilsen

Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011

ABSTRACT

Traditional dynamic memory management techniques for imperative programming languages are unsuitable for reliable real-time applications because their worst-case time and space requirements are insufficiently bounded. This is demonstrated by detailed measurements of several real-world workloads. A special hardware-assisted real-time garbage collection system has been designed to facilitate reliable use of dynamic memory in hard real-time systems. By analyzing the dynamic memory use of application software, the real-time developer can prove compliance with time and space constraints. Analysis techniques are presented and the real-time performance of the hardware-assisted garbage collection system is compared to that of traditional allocators.

2 May 1994

Reliable General Purpose Dynamic Memory Management for Real-Time Systems

Hong Gao
Kelvin D. Nilsen

Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011

1. Motivation

Dynamic memory management serves a key role in traditional software systems. Dynamic management of memory allows memory to serve different needs during different times depending on system workloads and computational modes. The use of dynamic memory reduces system costs, because common memory resources can be shared between multiple components. Dynamic memory also facilitates sharing of information between modules. Often, one module allocates a segment of memory and initializes its contents; then it passes the address of the shared memory segment to other program components. Thereafter, the shared memory segment serves as a buffer through which any one of the components that shares access to the common segment can broadcast messages to the other components that monitor the shared segment's contents. Finally, dynamic memory serves to significantly improve software functionality. In traditional software systems, implementations of friendly user interfaces and many important data processing algorithms make extensive use of dynamic memory management. To ignore the potential benefits of dynamic memory management in real-time environments is to seriously hobble real-time system designers and developers.

The standard interface to dynamic memory management in imperative languages such as C, C++, Pascal, and Ada consists of built-in allocate and deallocate operators. In C++, the allocate operator is called new and the deallocate operator is called delete. The new and delete operators invoke type-specific constructor and destructor functions when allocating and reclaiming dynamic memory respectively. C++'s new and delete implementations are generally implemented using C's malloc and free functions, for which numerous alternative implementations exist [1, 2]. In general, the various implementations offer a range of performance, with the most space efficient implementations running quite a bit slower than implementations that are more extravagant in their use of memory. Though most of the available implementations offer reasonably good average-case performance, none offers tight worst-case bounds on the time required to execute each malloc or free invocation. Furthermore, all existing implementations of malloc and free potentially suffer from memory fragmentation. In a fragmented memory system, free memory is scattered throughout the address space. Even though the sum of the sizes of individual free segments may represent a significantly large fraction of the entire memory heap, the system can honor subsequent memory allocation requests only if it can find sufficiently large segments of contiguous free memory.

Several alternatives are available to the real-time developer who requires dynamic memory management. Typically, the developer must create her own collection of subroutines to manage memory. At startup, these routines divide available memory into free blocks of several standard sizes. The system maintains a separate linked list of available memory blocks for each standard size. Each allocation request is served from the free list corresponding to the smallest standard-sized blocks that are at least as large as the requested memory allocation. This provides constant-time response to each allocation and deallocation request. However, memory utilization may be very poor. Each block of memory is permanently dedicated to representing objects of a particular size. It is not possible (within the requirements of constant-time response) to coalesce or divide allocated objects to meet changing dynamic memory needs. Thus, the startup code for the memory manager must anticipate the worst-case number of objects required for each of the standard object sizes in order to set aside the appropriate amount of memory for each

---

Draft Copy - Do Not Redistribute: revision dated May 2, 1994. This paper has been submitted to the 1994 Real-Time System Symposium.

of the free lists. Furthermore, because memory allocation requests may not exactly match the standard object sizes, much of the memory reserved within each dynamically allocated object is likely to go unused. Nevertheless, this dynamic memory management approach is very useful in real-time applications that are able to tolerate its limitations.

Often, considerable programmer effort is required to recognize the end of a dynamically allocated object's useful lifetime in order to recycle its memory. This chore is particularly difficult when many different program components share pointers to a common block of dynamic memory. In this case, the last component to destroy its pointer should take responsibility for reclaiming the object's memory. However, the order in which program components overwrite their pointers to shared objects likely depends on a variety of dynamic factors. Therefore, programmers must implement some form of dynamic bookkeeping to determine when it is safe to reclaim the memory allocated to particular dynamic objects. Reference counting [3] is the most commonly used technique for automatic reclamation of dead heap memory in real-time systems. By adding to each dynamic object an integer field that counts the number of pointers that refer to the object, the system can automatically reclaim the memory allocated to certain objects as a side effect of pointer assignments. Every time a pointer to a particular object is copied, that object's reference count is incremented by one. Every time a pointer is overwritten with a new value, the reference count field of the object originally referenced by the pointer is decremented by one. Though there is a certain appeal to automatic recognition and handling of dead objects, the overhead of maintaining each object's reference count is prohibitive. Every assignment to a pointer variable must adjust the reference count fields of two objects. Furthermore, whenever an assignment decrements a reference count field to zero, the system must place the corresponding object onto the appropriate free list.

Note that the reclaimed object may contain pointers to other objects, and these pointers must eventually be destroyed in order to decrement the reference counts of the objects they reference. In traditional reference counting systems, the act of placing an object onto the appropriate free list automatically overwrites all of the pointers contained within that object with zero, so that referenced objects can also be reclaimed. However, there is no bound on the length of the chain that might connect objects that are ready to be reclaimed. Thus, the time required to place an object onto the free list in traditional reference counting systems is unbounded. In order to support real-time constraints, pointers contained within free objects are overwritten with zero when the object is reallocated instead of when the object is reclaimed [4]. Therefore, the worst-case time required to perform a pointer assignment is the time required to adjust two reference counts and to link a dead object onto the appropriate free list. And the worst-case time required to allocate an object from its free list is the time required to overwrite each of the newly allocated object's pointers. Note that memory fragmentation is even more of a problem in reference-counting systems than in explicitly managed dynamic memory systems. If objects of one size contain pointers to objects of a different size, certain dead objects may be hidden beneath several levels of pointer indirection. Another problem with reference count systems is that the reference counting technique is not able to reclaim the memory associated with islands of objects that refer to themselves since the reference counts for these objects are always greater than zero. Thus, it is especially difficult to guarantee availability of memory in real-time reference counting systems.

2. Traditional Explicit Memory Management Techniques

All of the imperative programming languages in common use today provide dynamic management functions that are under direct control of the programmer. We have selected C++ as a representative language, and four popular C and C++ dynamic-memory allocators as representative implementations of the allocate and deallocate operations. We summarize the implementations of each of these allocators below. Note that our analysis focuses on general-purpose allocators for a general-purpose language. We address the question of whether it is appropriate to use these allocators in hard real-time applications.

SUN:

This is the allocator that is provided in the standard C libraries for SunOS 4.0. The free pool is represented by a Cartesian binary tree, in which nodes are ordered by ascending addresses (thus minimizing free list insertion time) and block sizes decrease with depth in the tree (thus minimizing search time for a block of a given size). When malloc is called, the search for a free block begins at the root of the tree. At each node, malloc asks which of its descendant nodes represents a better fit for the requested allocation. The best fit is the smallest node that is large enough to satisfy the request. The search stops when both descendants of the current node are too small. At this point, the current node is split in two. The first part is sized according to the user's request and its address is returned to the user. The second part comprises the remainder of the original object;

it is returned to the free pool. The heap is expanded if the user requests a block that is bigger than any block in the free pool. Given the address of an object to be deallocated, the free routine looks for neighbors with which to coalesce the object by searching the tree, starting from the root. If no neighbors are found, the deallocated object is simply added to the tree at the appropriate location.

GNU/C:

This allocator is distributed as part of the GNU libc distribution. This allocator uses two different allocation algorithms, depending on the size of the requested memory. For allocation of an object larger than the page size, the allocator uses a first-fit strategy, similar to the technique used for the KNUTH allocator described below. Smaller objects are allocated from page-sized chunks, within which all objects are the same size. A six-word chunk header describes the size of all objects contained within the chunk, maintains a count of the number of free objects within the chunk, and maintains a doubly-linked list of all the free free objects within the chunk. By masking off the least significant bits of an object's address, it is straightforward to find the corresponding chunk header. When malloc is called with the requested size, the size is rounded to a specific size class (a power of two). The allocator then traverses the chunk headers to search for the chunk that has at least one free block of the appropriate size. If no objects of a particular size class are available, more storage is allocated. When free is called to release an object smaller than the page size, the allocator determines which chunk contains the object to be freed and then links the block to the free list for that chunk. When freeing objects larger than the page size, the deallocator searches for neighboring free storage in order to coalesce the object if possible.

GNU/C++:

This allocator is distributed as part of the GNU C++ library. This algorithm, implemented by Doug Lea, enhances the standard first-fit algorithm by using an array of free lists. The i^{th} free list in the array keeps blocks of size $2^{(i+2)}$ to $2^{(i+3)}$, where $i = 0, 1, 2, \ldots, 29$. We call i the size index. In each free list, free blocks are connected in a doubly linked list. When malloc is called with the requested size t, the allocator rounds the size to the nearest multiple of 8 bytes, then takes the base two log of the result to get the size index i of the requested block. Then the i^{th} free list in the array is scanned for the first free block that is large enough to satisfy the request. This block is split into two blocks, one of the requested size that is returned to the application, and the other that is placed back on the appropriate free list. If no block is found, the allocator asks the operating system for more blocks with index size i. The heap is expanded at least 2K at a time. The newly allocated segment is divided into blocks of size $2^{(i+3)}$ and linked onto the i^{th} free list. When an object is freed, the size index is calculated and the block is linked onto the appropriate free list.

KNUTH:

This allocator, implemented by Mark Moraes, uses the first-fit strategy described by Knuth [5]. In this algorithm, free blocks are connected together in a doubly-linked free list that is scanned during allocation for the first free block that is sufficiently large to satisfy an allocation request. This block is split into two blocks, one of the appropriate size that is returned to the user, and the other that is placed back onto the free list. If the free portion is smaller than a threshold constant, the block is not split. The free list pointer is implemented as a ''roving'' pointer. This eliminates the aggregation of small blocks at the front of the free list. When deallocating objects, the free routine looks for neighbors with which to coalesce the object by searching the entire free list.

All of the algorithms reserve a small amount of memory to maintain bookkeeping information, including the size of each object and whether each object is currently free. In the GNU/C++, SUN, and KNUTH allocators, two extra words, called boundary tags, surround each allocated object. Boundary tags allow objects to be freed and coalesced with adjacent free storage in constant time. The GNU allocator uses a chunk header to contain information about the size of the blocks and status of the whole chunk as discussed in the description of the GNU allocator's implementation. Potentially, this reduces the space overhead associated with tagging, at the expense of a small amount of alignment padding within each chunk.

3. Garbage Collection

The term *garbage collection* describes the automated process of finding previously allocated memory that is no longer in use in order to make the memory available to satisfy subsequent allocation requests. Automatic garbage collection greatly simplifies the development effort required to manage dynamic memory. In systems that provide garbage collection, programmers need not concern themselves with explicit freeing of memory that is no longer in

use. Besides reducing the programmer's intellectual burden, this eliminates several common dynamic programming errors: the failure to free memory when it is no longer in use, freeing of the same block of memory multiple times, and accidental freeing of memory before its useful lifetime has ended. These sorts of programming errors are especially difficult to find and correct because the consequence of an error is generally not detected at the time the error occurs. Rather, the error manifests itself much later in program execution, usually in one of the following ways:

- The system runs out of memory because of an accumulation of failures to free unused memory.
- The system's dynamic memory manager becomes confused because objects accidently placed onto its free list are still being used. The bookkeeping information that the dynamic memory manager stores within objects on its free list is likely to become corrupted by continued use of the object.
- The dynamic memory manager's free pool becomes corrupted because an attempt was made to insert a particular object onto its free list more than once, without ever having removed the object from the free list.
- The application becomes confused because the dynamic memory manager reallocates an object that was erroneously placed onto its free list. Thereafter, the memory serves two different purposes.

Rovner [6] estimates that the programming effort required to perform dynamic memory management is approximately 40% of the total cost of developing a large software system. This includes both the costs of developing and of debugging the memory management routines.

Besides reducing the complexity of dynamic memory management, some modern garbage collection algorithms offer storage throughputs that far exceed the capabilities of traditional memory management techniques. For example, the time required by the system described in this paper to allocate and reclaim dynamic memory has been measured as approximately one fifth the time required to manage the same total amount of dynamic memory using traditional C++ heap management techniques.

Traditional garbage collector implementations periodically suspend application processing in order to traverse all of memory in search of segments that are no longer in use. The processing delay associated with occasional garbage collections is inconvenient in interactive applications and unacceptable in real-time environments. Incremental garbage collectors allow application processing to continue while garbage collection is performed. But the frequent synchronization that is required between background garbage collection activities and ongoing application processing significantly reduces system throughput [7, 8].

One other significant overhead associated with most garbage-collection systems is the effort required to tag data stored within the garbage-collected heap so that the garbage collector can traverse live data structures[1]. In dynamically typed languages, like Smalltalk [9] and Icon [10], all data structures are generally tagged to support dynamic type checking, so the tags required for garbage collection are available without any additional run-time overhead. However, in statically typed languages such as C++, the burden of tagging data using naive techniques in order to support garbage collection has been measured to nearly double the execution time of many real programs [11]. For a combination of these reasons, many developers feel that garbage collection is a luxury they simply cannot afford.

Real-time garbage collectors must honor tight upper bounds on the duration of time during which they might suspend execution of application processing. In existing systems, these delays are imposed during reading and writing of heap-allocated memory, and during allocation of new objects. It is important for these operations to be time-bounded so that the worst-case time required to execute real-time tasks that invoke these operations can be determined through static analysis. Though a number of incremental garbage collection techniques have been developed [8, 12-17], the best worst-case latencies available from garbage collectors built on stock hardware (without special hardware assists such as the ones described in this paper) range from 500 μsec to several msec [18-20]. Not only do these software-only real-time garbage collectors suffer from worse latencies than our system, but they are also much less general in terms of the types and sizes of objects that can be automatically managed by the hardware-assisted garbage collection system.

In this paper, we describe an incremental C++ garbage collector that has low synchronization overhead, minimal pointer tagging overhead, and guarantees to complete all memory fetch, store, and allocate operations in less than 1 μsec. The system makes use of special hardware placed within an expansion memory module to achieve high throughput and tight worst-case bounds on the time required to perform particular operations. If the special

[1] Conservative garbage collectors avoid the cost of data tagging by assuming that every memory word and register that contains a value representing a legal address is, in fact, a pointer to an object residing in memory.

hardware is mass produced, we estimate that the cost of a garbage-collecting memory module (GCMM) is three to five times the cost of the memory required to represent the same total amount of live data, assuming that the data is perfectly packed[2]. But it is rare in practice for high-performance memory managers to achieve 100% utilization of memory. Rather, typical memory utilizations range from 17 to 85% [1]. Cost comparisons between the hardware-assisted memory manager and traditional memory management techniques must take these factors into account.

Our system garbage collects all of the objects allocated within the C++ dynamic heap without requiring any changes to C++ syntax. There are a few C++ practices that must be avoided in order for the garbage collector to operate correctly[3]:

1. Pointers should not be coerced to integers. A common practice is to use object addresses as hash values. Since our garbage collector relocates objects to eliminate fragmentation, object addresses are not constant.
2. Integers should not be coerced to pointers. Whenever the garbage collector relocates an object, it automatically updates all pointers that refer to that object so that the pointers refer to the object's new location. Any pointers hidden within variables declared as integers will not be updated.
3. All pointers need to refer to addresses contained within the objects they point to. In case a pointer refers to an array, the pointer may point to an imaginary element appended to the end of the allocated array. (This restriction is already specified in the C++ standard; we mention it here for emphasis.)
4. Every assignment to a union field that represents both pointer and non-pointer data must be visible to the compiler as a union field assignment. Assignments by way of a pointer to the union field violate this constraint.
5. The internal organization of every heap-allocated object must be represented by the argument to new.

Our experience porting existing C++ code to the garbage-collected C++ implementation reveals that most existing code already complies with these restrictions. For example, the troff component of James Clark's groff implementation is compiled from over 23,000 lines of C++ code. To make this compatible with our garbage collector, we rewrote only four lines. In each case, the original code new-allocated an object as an array of characters and coerced the resulting address to a structure pointer, violating the fifth constraint listed above. We have also ported several C applications to the garbage-collected environment. In general, we found that retargeting C code was more difficult than retargeting C++ code. See reference 22 for an explanation.

Space does not allow for a complete description of the garbage collection algorithm and its hardware-assisted implementation. The system is described more thoroughly in references 22 and 23. Below, we discuss some of the real-time considerations that govern use of the garbage collector.

The Garbage Collection Flip

The garbage collection algorithm periodically copies all live objects from one region of memory, named *from-space* to another region of memory, named *to-space*. At the start of each garbage collection, the names given to the two equal-sized spaces are exchanged. We call this a garbage collection flip. While garbage collection is active, new objects can be allocated starting from the end of *to-space* while old objects are being copied into the beginning locations of *to-space*.

If the CPU uses a write-back cache, it is necessary at the time of a flip to synchronize the cache with memory at the time of a flip. Potentially, the time required to write the entire cache back to memory is longer than can be tolerated in certain real-time environments. For example, the time required to write 16 KBytes of cached data to memory on a typical embedded processor is likely to range between 100 and 500 μs. If the CPU uses a write-through cache, then it is only necessary to invalidate cached heap data, without writing it back to memory. This can

[2] We have recently redesigned the hardware-assisted memory module so that it supports a hybrid garbage collection algorithm that collects garbage in certain regions using incremental mark-and-sweep methods and collects garbage in other regions using real-time copying techniques [21]. Though we have not yet measured this new algorithm's throughput, we expect that overall performance will be roughly equivalent, if not superior, to the fully copying real-time garbage collection technique that we have already measured. The main motivation for this alternative garbage collection system is to reduce the amount of real memory required to support a particular application's dynamic memory needs. Whereas the original design costs three to five times more than the cost of the perfectly packed memory required to represent an application's worst-case memory needs, the revised design is likely to cost only 20 to 50% more than the perfectly packed memory.

[3] View these requirements as principles of operation rather than hard-and-fast rules. There are many cases in which, for example, it is perfectly reasonable to coerce between integer and pointer types. In these cases, it is the programmer's responsibility to verify that the code does not violate the integrity of the garbage collector.

generally be achieved in much less time than is required to write the cached data back to memory. In either case, analysis of the real-time workload must account for the occasional delay imposed by the flip operation.

Flips are always separated by the time required to perform a complete garbage collection. With typical workloads, the time between flips is no less than 10 seconds [23]. By analyzing the dynamic memory allocation behavior of an application, as discussed below, it is possible to guarantee a minimum time separation between flips.

Pacing of Allocation

In order for garbage collection to operate reliably, it is very important that the total amount of copied memory combined with the total amount of memory allocated while garbage collection is active is less than the total size of *to-space*. Suppose that the size of each space, measured in bytes, is represented by M, and the worst-case amount of live memory retained by the application is $N < M$. At the time garbage collection begins, no more than N bytes of live memory exists in *from-space*. It is likely that some of this memory will become dead before the garbage collector manages to copy it. But to be safe, assume that the garbage collector copies all N bytes into *to-space*. Then, the application must allocate no more than $A = M - N$ bytes during the time that garbage collection is active. Note that it is very important that no more than A bytes of memory be allocated during garbage collection, and that the application retain no more than a total of N bytes of live memory. If, for example, new memory allocations consume part of the region into which old live objects are to be copied, then the garbage collector would need to discard certain live objects from the heap in order to continue execution.

Let U represent the total amount of time required to complete garbage collection, in the worst case. The following techniques are available to real-time developers who want to ensure that garbage collection operates reliably.

1. Assume nothing about the dynamic memory needs of the application. In this case, it is not possible to bound the time required to allocate memory. However, it is possible to guarantee that previously allocated objects will continue to be accessible. At the start of garbage collection, assume that all of the memory allocated within *from-space* is still live. Recompute A for each garbage collection pass. While garbage collection is active, refrain from allocating a total of more than A bytes of new memory. Once garbage collection completes, you will generally discover that much of the memory that had been allocated in *from-space* did not need to be copied into *to-space*. So the memory originally reserved for the copying of these dead objects can now be allocated to serve ongoing application needs.
2. Assume that the application is known never to retain more than N bytes of live memory. In general, a total of more than N bytes will have been allocated within *from-space* at the time that garbage collection begins. But the garbage collector is guaranteed that no more than N of these bytes need to be copied. Make sure that no more than A bytes are allocated while garbage collection is active. The main difference between this technique and option 1 is that A is constant for this technique. There are several ways to enforce this constraint.
    2a. Allocate all memory requests as quickly as possible, as long as the total amount of allocated memory is less than A. But don't honor requests (stall or simply fail) that would require allocation of more than A bytes.
    2b. For each allocation request, honor the request only if the fraction of garbage collection completed is greater than the total amount of allocated memory divided by A. Otherwise, stall or fail in response to the allocation request.
    2c. Same as 2b, except do not require equal amounts of garbage collection to accompany equal amounts of allocation. Instead, allow greater amounts of allocation during the early phases of garbage collection, and lesser amounts during the later phases. The total amount of allocation permitted during garbage collection is the same, but this technique allows improved average-case performance by reducing the need to stall the application [24]. This technique is described in greater detail in reference 23.
3. Assume that the application is known never to retain more than N bytes of live memory, and that the application is known never to allocate a total of more than A bytes within any time period U. Under this assumption, average-case throughput is improved because the allocator does not need to monitor the progress of the garbage collector. And worst-case latencies are improved because it is never necessary to stall the CPU's allocation requests. The performance measurements reported in section 5 of this paper use this pacing technique.

Though programmers of traditional non-real-time applications are not accustomed to doing the sorts of analysis that would be required to prove that a particular application interacts with the garbage collector in a reliable way, real-

time programmers are accustomed to taking great care to specify and verify that the resource needs of particular program components are well understood and tightly bounded.

4. Experiments

To compare the throughput and latencies of the garbage collection system with traditional explicit dynamic memory management techniques, we have analyzed a number of experimental workloads (benchmarks) using both the garbage-collected and explicitly managed dynamic memory heaps. We have modified the dlxsim architectural simulator for the hypothetical DLX architecture [25] so that it simulates instruction and data caches, and a GCMM. Additionally, we have modified an implementation of the GNU C++ compiler, version 1.31.1, so that it generates code that communicates with the GCMM for all dynamic memory management. Each of the benchmark workloads was compiled with the traditional GNU C++ compiler and linked with the four different memory allocators. Each workload was also compiled with the custom GNU C++ compiler that generates code to interact with the GCMM. The six experimental workloads are described briefly below:

cham:
: Chameleon is an N-level channel router for multi-level printed circuit design. The ex2 file, distributed with the release code, was used as input to the program.

espr:
: Espresso is a logic optimization program. The Z5xp1 file, distributed with the release code, was used as input to the program.

cfrac:
: Cfrac is a program to factor large integers using the continued fraction method. Input was the 22-digit number 1,000,000,001,930,000,000,057; which is the product of two primes.

sfft: Sfft performs a sliding discrete Fourier transform on a file of 8-bit audio data. The file *medium*, distributed with this benchmark application, was used as input to the program.

lisp: This is a C++ implementation of a lisp interpreter, written by Tim Budd. The file prune2.lsp, distributed with this application, was the input file to the program.

troff:
: This is James Clark's troff program, written in C++, from the GNU groff typesetting package. The file *osm-paper2*, distributed with this benchmark application, was used as input to the program.

These programs represent a range of memory intensive tasks with different execution behaviors.

The original C versions of the cham, espr and cfrac programs are publically available via anonymous FTP from ftp.cs.colorado.edu in the directory /pub/cs/misc/malloc-benchmarks. The modified C++ versions of all of the experimental workloads are available from ftp.cs.iastate.edu in the directory /pub/kelvin/malloc-benchmarks.

5. Performance Measurements

Table 5.1 summarizes the relevant performance characteristics of the traditional implementations of each experimental workload, as measured when linked with the GNU/C++ allocator. Note that these applications vary widely in their behavior. sfft, for example, only allocates 2 objects, each of which is an I/O buffer. cham and lisp allocate a large number of objects, without freeing most of them. espr and troff both allocate and deallocate a large number of objects. cfrac allocates fewer objects, and deallocates most of the objects that it has allocated. The sixth column of Table 5.1 reports the number of instructions executed, on average, for each function invocation. This gives an indication of the frequency with which function invocations take place. The protocol for managing function activation frames is more costly in the garbage-collected C++ implementations of each workload. This factor is especially noticeable for the cfrac, troff, and lisp workloads. The seventh column of Table 5.2 reports the number of instructions executed per allocated byte. Note that the applications with the smallest instructions per byte allocated are the ones for which the cost of dynamic memory management is the highest percentage of total execution time (see column 8).

Table 5.1: Workload characteristics							
Benchmark	# Alloc	# Free	Largest Alloc (bytes)	Total Alloc (bytes)	Instructions per Call	Instructions per Byte Allocated	Alloc/Free Time (% of total)
cham	19,776	60	38,420	748,325	132	77.9	8.2
espr	24,743	24,270	8,200	1,061,696	192	50.4	10.3
cfrac	1,364	1,054	8,200	184,104	58	702.9	0.5
troff	49,644	35,107	20,480	1,411,465	45	221.4	4.8
lisp	38,839	0	8,200	646,507	23	387.6	3.6
sfft	2	0	8,200	16,400	448	14,835	0.003

We instrumented the dynamic memory management routines so that we could measure how much time was spent allocating new memory. The worst-case costs are summarized in Table 5.2. Note that the costs of allocating new objects using the garbage-collected C++ implementations are all less than 110 machine cycles, whereas the worst-case costs of allocating new objects using the traditional memory allocators range from 2,000 to over 150,000 machine cycles. Clearly, use of traditional memory allocators in real-time applications represents a potential source of unpredictable timing performance.

It is important to recognize that Table 5.2 reports worst-case measured execution times. For most of the explicit memory management techniques, it is not practical to analytically derive the worst-case times required to service allocation requests. In contrast, proofs of real-time performance have been demonstrated for the garbage-collector's allocator [23].

Table 5.2: Worst-Case Allocation Delays (Machine Cycles)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	7,285	4,335	139,804	2,586	81
espr	11,615	4,335	142,957	5,505	101
cfrac	4,952	5,109	141,901	2,382	87
troff	7,336	5,867	142,729	6,428	104
lisp	4,357	4,353	151,576	2,873	107
sfft	5,745	5,745	5,745	4,740	48

For the same workloads, Table 5.3 summarizes the average delays associated with allocation of new memory. Note the order of magnitude differences between the values reported in tables 5.2 and 5.3. Note also that the average cost of allocating memory in the garbage-collected system is typically less than 10% of the cost of allocating memory using the best traditional techniques. We consider the sfft application not to be statistically significant in this measure, because it only allocates two relatively large objects.

Table 5.3: Average Allocation Delays (Machine Cycles)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	195.3	271.4	327.8	258.6	17.3
espr	152.0	359.9	296.3	463.3	15.8
cfrac	384.6	775.2	959.8	213.1	16.2
troff	246.3	644.2	358.7	224.1	18.8
lisp	233.3	331.2	375.4	200.9	18.1
sfft	3,061.5	3,061.5	3,061.5	2,559.0	28.5

Tables 5.4 and 5.5 report the worst- and average-case costs of deallocating memory. Note that deallocation is free in the garbage-collected system.

Table 5.4: Worst-Case Deallocation Delays (Machine Cycles)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	506	986	3,767	839	0
espr	842	2,205	7,796	924	0
cfrac	771	2,001	9,837	948	0
troff	828	4,533	8,866	1,872	0

Remember that the cost of deallocating an object includes the cost of searching for neighboring objects with which to coalesce the deallocated object. This is why, for several of the allocators, the costs of deallocating objects is even higher than the costs of allocating them.

Table 5.5: Average Deallocation Delays (Machine Cycles)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	90.7	563.5	475.8	215.5	0
espr	71.7	598	287.3	187.4	0
cfrac	79.3	770.2	277.8	206.5	0
troff	82.4	815.0	302.0	283.7	0

Memory is a costly component in modern computer systems. In many of tomorrow's embedded computers, memory will be the single most expensive component in the system. To achieve high memory utilization is one of the goals of a system designer. We have measured the memory heap size and memory usage of the traditional allocators and the garbage collection system. Heap sizes are reported in Table 5.6. The traditional allocators start out with a small heap, and expand the heap as necessary in order to support the application. The garbage-collected allocator uses a fixed size heap throughout execution of each workload. The reported heap size, which includes both *from-space* and *to-space*, is constrained by limitations in our current simulator to be a power of two. For most of the applications, the GC heap is much larger than the traditional heap. This is because the GC heap is unable to utilize more than one of its two semispaces at a time. For the lisp application, the GC heap is much smaller than the traditional heap. For cham, the GC heap is approximately the same size as the traditional heaps. In both cases, the garbage collector is able to reclaim memory that the programmers failed to deallocate.

Table 5.6: Maximum Heap Size (bytes)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	1,089,600	983,048	1,001,760	942,722	1,028,576
espr	152,000	49,160	95,824	61,467	262,144
cfrac	241,968	196,616	259,952	204,975	1,028,576
troff	760,096	532,488	508,768	680,426	2,097,152
lisp	1,266,208	1,007,624	761,536	946,857	262,144
sfft	37,392	37,264	37,296	37,488	262,144

We define heap utilization as the quotient of the combined sizes of all live objects divided by the heap size. For the traditional implementations, live objects are defined as objects allocated but not yet freed, and heap size is the maximum size of the sbrk region. For the garbage-collected implementations, live objects are defined as all of the objects in existence immediately following completion of garbage collection, and the heap size is defined as the combined size of both semispaces. For both the garbage-collected and traditional allocators, the size of a live object is defined to include whatever tags accompany it. Table 5.7 reports the average heap utilization of the various workloads. Note that the utilization of traditional allocators is not directly comparable with that of the garbage-collected implementations because of the different definitions of utilization.

Table 5.7: Percent Memory Utilization (Average)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	45	50	43	53	35
espr	16	44	26	43	15
cfrac	36	48	34	46	28
troff	64	90	79	71	23
lisp	38	49	48	51	25
sfft	32	33	32	32	*no data*

Table 5.7 must be interpreted carefully. Note, for example, that lisp's traditional allocators report much higher utilization than the garbage collector, even though the garbage collector executes in a smaller heap. This is because of the differences in the definition of liveness between traditional and garbage-collected allocators. In an attempt to calibrate these figures, we have quantified the storage leaks that are present in each of the applications, as tabulated in Table 5.8.

Table 5.8: Storage Leaks (estimated)	
Benchmark	Percent Leak
cham	45
espr	20
cfrac	22
troff	5
lisp	90

The percentages reported in Table 5.8 were determined by comparing the amounts of live memory following termination of the last garbage collection performed during each of the benchmarks. In the cham application, for example, there were X bytes of live memory at the time that the last garbage collection completed. After the GNU/C++ implementation of cham had completed approximately the same fraction of its computation as the garbage-collected implementation had completed when we last measured its live memory, the difference between the amount of memory that had been allocated and the amount of memory that had been deallocated was Y bytes, where $(1 - 0.45) \times Y = X$. Note that in all cases, the garbage collector's notion of live memory is smaller than that of the traditional allocators.

Table 5.9: Adjusted Percent Memory Utilization (Average)					
Benchmark	GNU/C++	SUN	GNU	KNUTH	GC
cham	25	27.5	24	29	35
espr	13	35	21	34	15
cfrac	28	37	27	36	28
troff	61	86	75	67	23
lisp	4	5	5	5	25

Table 5.9 reports adjusted memory utilizations for each of the allocators. This table was computed by adjusting the traditional allocator utilizations by the estimated storage leak percentages reported in Table 5.8. Note that Table 5.9 correlates roughly, but not exactly, with Table 5.6. The main differences between Table 5.6 and 5.9 result from the differences in the memory utilization sampling frequencies. For the traditional allocators, the average utilization was computed by averaging the utilizations at the point of each allocation, whereas for the garbage-collected allocator, the average utilization was computed by averaging the utilizations at the end of each garbage collection. For most of the test cases, there were thousands of allocations and fewer than ten garbage collections. All of the statistics reported in tables 5.7 through 5.9 should be viewed with this in mind. The main point to be emphasized is that the GC implementations generally offer much poorer memory utilization than the traditional implementations.

Although worst-case latencies and memory utilization are the primary concern of real-time developers, a cost-effective real-time system must also exhibit good average-case performance. Table 5.10 reports the total execution times for each of the workloads, measured in machine cycles.

Table 5.10: Execution time (10^6 cycles), Instruction count (10^6 instructions), CPI						
Benchmark	quantity	GNU/C++	SUN	GNU	KNUTH	GC
cham	Exec. Time	73.6	75.6	76.2	78.1	79.0
	Instr. Count	58.3	60.5	60.9	58.2	57.3
	CPI	1.263	1.250	1.252	1.302	1.379
espr	Exec. Time	67.5	88.4	78.9	79.0	70.9
	Instr. Count	53.5	67.8	59.6	60.7	53.7
	CPI	1.260	1.304	1.324	1.302	1.320
cfrac	Exec. Time	151.1	154.2	152.9	150.3	193.0
	Instr. Count	129.4	130.1	129.0	129.5	138.6
	CPI	1.168	1.185	1.185	1.161	1.392
troff	Exec. Time	519.2	627.1	599.0	557.1	512.5
	Instr. Count	312.5	338.6	322.4	316.4	335.7
	CPI	1.662	1.852	1.858	1.761	1.527
lisp	Exec. Time	304.3	308.6	313.8	298.5	380.8
	Instr. Count	250.6	254.1	255.5	251.1	304.1
	CPI	1.214	1.215	1.228	1.189	1.252
sfft	Exec. Time	309.6	309.0	309.0	309.1	313.4
	Instr. Count	243.3	243.3	243.3	243.3	244.5
	CPI	1.273	1.270	1.270	1.270	1.282

As discussed above, memory allocation and deallocation runs much faster in the GC implementation than in the traditional systems. Programs for which the costs of memory management are relatively large (5% or higher) are most likely to benefit from this performance advantage. Programs that do not deallocate memory do not benefit as much from this factor as those that do. The GC system imposes a 9-instruction overhead on each function call. Thus, applications that require frequent function calls generally suffer worse performance in their garbage-collected implementations. This is especially noticeable in the cfrac and lisp measurements. This effect is counterbalanced by other effects in the troff benchmark, as discussed below.

From examination of Table 5.10, it is clear that automatic garbage collection of C++ performs competitively with traditional techniques for dynamic memory management. A thorough analysis of the factors that influence performance is beyond the scope of this paper. See reference 26 for additional explanation. Below, we summarize our analysis of the measured performance:

- Of the measured workloads, sfft is the only one that does not make extensive use of dynamic memory, requesting only two memory allocations during execution. This workload helps characterize the overhead of the garbage collection system on applications that don't benefit from the use of dynamic memory management. In sfft, the overhead of communicating with the garbage collector is minimal--- slightly over 1% in execution time and about 0.5% in instruction count when compared with the four traditional allocators.
- The lisp application has the smallest number of instructions executed per function call, and never frees allocated memory. This application demonstrates the worst measured performance for the garbage collector. The garbage-collected implementation exhibits 20% to 27% overhead in execution time and 20% overhead in instruction count when compared with the traditional allocators.
- cfrac is another program that has a relatively small number of instructions executed per function invocation. Unlike lisp, cfrac does free most of its dynamic memory. However, cfrac spends only 0.5% of its execution time managing dynamic memory, so the improved efficiency of the garbage collector's memory manager has little to offer this application. The main difference between cfrac's traditional implementations and its garbage-collected implementation is the increase in instructions required to implement function entry and exit in the garbage-collected version. A second difference between the traditional and garbage-collected implementations is that the garbage-collected system exhibits much poorer cache hit rates. This is because each flip, which occurs on average once every 1.8 million cycles in this particular application, invalidates all of the cache lines that hold heap-allocated data[4].

[4] The high frequency of flips results from the relatively small semispace size (512 Kbytes). For better performance,

- The cham program executes many memory allocations with few deallocations, which means that the traditional implementations do not incur the overhead of freeing objects. Similar to lisp, this inflates the overhead of the garbage collector. However, cham has larger functions (more executed instructions per function activation frame) than lisp. Thus the overhead of managing function entry and exit is not as large. Given this, the garbage-collected cham executes only slightly slower than the traditional implementations.
- In espr and troff, the garbage collector runs more efficiently than the traditional implementations, about 20% faster than the SUN allocator, 15% faster than the GNU allocator and 10% faster than the KNUTH allocator. The GNU/C++ allocator runs in approximately the same time as the garbage collector. For espr, the improvement seems to result mainly from the improved efficiency of the garbage-collector's memory management. Note that espr spends a higher percentage of its execution time managing dynamic memory than any of the other experimental workloads. For troff, this improvement in performance is primarily due to better interleaving of memory access, since the garbage-collection algorithm takes better advantage of memory units that have been optimized for sequential access. The traditional implementation uses a single bank of memory to represent all code and data. The garbage-collected implementation uses one bank of memory to represent code and static data known not to contain pointers and a different bank of memory to represent the dynamic heap. Since each bank of memory is implemented using static-column DRAMS, localized memory accesses within each bank perform better than completely random access. By separating the dynamic heap and code into distinct memory banks, the garbage-collected implementation improves the locality of memory references within each bank.

6. Discussion

We have shown that traditional techniques for dynamic memory management are incompatible with real-time performance constraints. Furthermore, we have demonstrated that garbage collection of imperative languages such as C++ is feasible with minimal impact on existing language definitions. Finally, we have demonstrated that with special hardware support, garbage collection of high-performance real-time programs is also possible. The measurements reveal two major shortcomings of the garbage collection system:

1. Memory utilization is generally much worse in the garbage-collected implementation than in the traditional systems.
2. On average, the overhead of function entry and exit for the garbage-collected implementations of many applications is large.

Since memory is an expensive component of most real-time computing systems, it is important to improve memory utilization. We have recently designed a hybrid defragmenting real-time garbage collector that garbage collects one portion of memory using the copying technique described in this paper, and garbage collects the remaining memory using an incremental mark and sweep technique [21]. Unlike the current design, which can allocate at most one half of the total heap, the new design allows the allocator to allocate up to $\frac{(N-1)}{N}$ of the heap, where N is an integer number typically in the range from 2 to 16. Additionally, work is under way to develop an optimizing compiler that avoids the 9-instruction overhead of function invocation. See reference 24.

Additional study is required to investigate a wider variety of C++ applications, both to measure their performance on the experimental architecture and to determine the degree to which arbitrary C++ code complies with the special garbage collection requirements described in section 3 of this paper. Currently, we are working to develop a hardware prototype of the proposed memory architecture [21] and an optimizing C++ compiler for the target system. Availability of this prototype system will greatly expand the domain of real-time applications that we can effectively analyze.

7. Acknowledgements

We would like to thank Jim Lathrop and Craig Vanzante, students at Iowa State University, for developing the sfft and lisp test cases. William Schmidt implemented the C++ compiler support required to generate code that interacts with the garbage collection hardware. He also ported the groff software to the garbage-collected C++ environment. We thank Benjamin Zorn of the University of Colorado at Boulder for his help in acquiring the cham, espr, and cfrac applications and test data.

use a larger semispace size.

This work was supported by the National Science Foundation under grant MIP-9010412, and by a grant from the U. S. Department of Commerce.

References

1. B. Zorn and D. Grunwald, Evaluating Models of Memory Allocation, *ACM Transactions on Modeling and Computer Simulation 4*, 1 (Jan 1994), .
2. D. Grunwald and B. Zorn, CustoMalloc: Efficient Synthesized Memory Allocators, University of Colorado Tech. Rep. CU-CS-602-92, July 1992.
3. G. E. Collins, ‘‘A Method for Overlapping and Erasure of Lists’’, *Comm. ACM 3*, 12 (Dec 1960), 655-657.
4. J. Weizenbaum, ‘‘Symmetric List Processor’’, *Comm. ACM 6*, 9 (Sep. 1963), 524-544.
5. D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental algorithms*, Addison-Wesley, 1973.
6. P. Rovner, On Adding Garbage Collection and Runtime Types to a Strongly-Typed Statically-Checked, Concurrent Language, CSL-84-7, Xerox Palo Alto Research Center, 1984.
7. C. Chambers, Cost of Garbage Collection in the SELF System, *1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*, Phoenix, AZ, Oct 1991.
8. K. Nilsen, ‘‘Garbage Collection of Strings and Linked Data Structures in Real Time’’, *Software—Practice & Experience 18*, 7 (July 1988), 613-640.
9. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
10. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
11. W. J. Schmidt, Issues in the Design and Implementation of a Real-Time Garbage Collection Architecture, Ph.D. Dissertation, Iowa State Univ. Tech. Rep. 92-25, 1992.
12. H. G. Baker Jr., ‘‘List Processing in Real Time on a Serial Computer’’, *Comm. ACM 21*, 4 (Apr. 1978), 280-293.
13. D. Ungar, Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *SIGPLAN Notices 19*, 5 (May 1984), 157-167.
14. T. W. Christopher, ‘‘Reference Count Garbage Collection’’, *Software—Practice & Experience 14*(1984), 503-507.
15. H. Boehm, A. J. Demers and S. Shenker, ‘‘Mostly Parallel Garbage Collection’’, *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, June 1991.
16. T. Yuasa, Real-Time Garbage Collection on General-Purpose Machines, *Journal of Systems and Software 11*(1990), 181-198.
17. S. Nettles, J. O’Toole, D. Pierce and N. Haines, Replication-Based Incremental Copying Collection, in *Memory Management*, Y. Bekkers and J. Cohen (ed.), Springer-Verlag , 1992, 357-364.
18. S. L. Engelstad and J. E. Vandendorpe, Automatic Storage Management for Systems with Real-Time Constraints, *Oral presentation at 1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*, Phoenix, AZ, Oct 1991.
19. R. Johnson, Reducing the Latency of a Real-Time Garbage Collector, *ACM Letters on Prog. Lang. and Systems 1*, 1 (Mar 1992), 46-58.
20. J. R. Ellis, K. Li and A. W. Appel, ‘‘Real-time Concurrent Collection on Stock Multiprocessors’’, *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, June 1988.
21. K. Nilsen, Cost-Effective Hardware-Assisted Real-Time Garbage Collection, *ACM SIGPLAN Notices Workshop on Language, Compiler, and Tool Support for Real-Time Systems* , June 1994.
22. K. Nilsen, Reliable Real-Time Garbage Collection of C++, *Computing Systems 6*, 4 (Fall 1994), .
23. K. D. Nilsen and W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System, *Journal of Programming Languages*, To appear.

24. W. J. Schmidt and K. D. Nilsen, Performance of a Hardware-Assisted Real-Time Garbage Collector, *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Submitted.

25. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

26. H. Gao, Comparative Performance of a Hardware-Assisted Real-Time Garbage Collector for C++, Master's Thesis, 1994.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR94-09
Submission Date: July 14, 1994